Uncoupling Coordination: Tuple-based Models for Mobility

Giacomo Cabri, Luca Ferrari, Letizia Leonardi, Marco Mamei, Franco Zambonelli

Università di Modena e Reggio Emilia – ITALY

{cabri.giacomo, ferrari.luca, leonardi.letizia, mamei.marco, zambonelli.franco}@unimore.it

Abstract. This chapter focuses on tuple-based (Linda-like) coordination models as middleware services for mobile and pervasive computing systems. After having introduced the basic concepts of tuple-based coordination, the chapter discusses the suitability of tuple-based models for mobility and introduces a simple taxonomy of tuple-based middleware models for mobile systems. Then, on the basis of the introduced taxonomy, the chapter presents several proposals – both industrial and academic – that have been made in the area. Eventually, the paper outlines open research issues and promising research directions in the area of tuple-based coordination models for mobile computing systems.

TABLE OF CONTENTS

1 Introduction

Computing is becoming intrinsically mobile and ubiquitous [EstC02, CabLMZ03]. Computer-based systems are going to be embedded in all our everyday objects and in our everyday environments. These systems will be typically communication enabled, and capable of coordinating with each other in the context of complex distributed applications, e.g., to support our cooperative activities, to monitor and control our environments [Bor02], and to improve our interactions with the physical world [MamZ04]. Also, since most of the embeddings will be intrinsically mobile, as a car or a human, distributed software processes and components (from now on, we adopt the term "agents" to generically indicate the active components of a distributed application) will have to effectively interact with each other and effectively coordinate their activities in a context-aware way, despite the network and environmental dynamics induced by mobility. Identifying proper coordination models and the associated middleware services to effectively rule and control agents' activities is thus a key research issue.

Among several proposals, tuple-based models (rooted on the Linda coordination language [GelC92]) appear very suitable at supporting coordination activities in mobile computing settings. By promoting indirect coordination via a sort of shared dataspace, tuple-based coordination uncouples interacting agents and relieves them from the need of knowing a priori each other and of knowing their respective positions, information which would be otherwise costly to obtain in dynamic and mobile computing scenarios. Also, shared dataspace coordination models, such as tuple-based ones, naturally supports context-aware coordination models.

Starting from the basic suitability of tuple-based coordination to mobile computing, a number of diverse solutions can be conceived (and have been proposed) to actually promote tuple-based coordination in the form of middleware-level services for mobile computing. The aim of this chapter is to overview the space of such possible solutions, and to survey the most relevant proposal for tuple-based coordination in mobile computing systems. A specific focus will be given to the software engineering implications, i.e., to the analysis of the support that the surveyed models and middleware give to the software architect/programmer, in term of abstractions, tools, and APIs.

2 Tuple-based Coordination

Tuple-based coordination has been firstly introduced in the late 80's, in the form of the Linda coordination language for concurrent and parallel programming [GelC92], and consisted in a limited set of primitives (the *coordination primitives*) to access to a *tuple space*. Later in the 90's, the model got widespread recognition as a general-purpose coordination paradigm for distributed programming.

The atomic units of interaction in tuple-based coordination are tuples. A tuple is structured set of typed data items, i.e., a record. Coordination activities between application agents (there included synchronization) can take place indirectly via indirect exchange of tuples through a shared *tuple space*, i.e., a sort of shared dataspace that act simply as a tuple container. The coordination primitives provided to agents are a means to access and manipulate a shared tuple space.

A tuple can be written in the tuple space by an agent performing the `out` output primitive. As an example, `out("amount", 10, a)` writes a tuple with three fields: the string `amount`, the integer `10` and the contents of the program variable `a`. Two input primitives are provided to associatively retrieve data from the tuple space, `rd` and `in`, to read or extract, respectively, a tuple from the tuple space.

A *matching rule* governs tuple selection to retrieve tuples from a tuple space in an associative way: input operations take a *template* as their argument, and the returned tuple is one *matching* the template. In order to match, the template and the tuple must be of the same length, the field types must be the same, and the values of constant fields have to be identical. For instance, the operation `in("amount", 10, ?b)` looks for a tuple containing the string amount as its first field, followed by the integer `10`, followed by a value of the same type as the program variable `b`: the notation `?b` indicates that the retrieved value is to be bound to the variable `b` after retrieval. Input operations are *blocking*, that is, they return only when a matching tuple is found, thus providing a mechanisms for two agents to indirectly synchronized based on tuples' occurrences. When multiple tuples matches a template, one is selected non-deterministically.

Other Linda operations include `inp`, `rdp` – the *predicative*, *non-blocking* versions of `in` and `rd`, which return true if a matching tuple has been found and false otherwise – and `eval`, which is intended to create an active tuple, i.e., a tuple where one or more fields do not have a definite value, but must be computed by function calls. When such a tuple is emitted, a new process is created for each function call to be computed. Eventually, when all these processes have performed their computation, the active tuple is replaced by a regular (passive) tuple, whose function calls are replaced by the corresponding computed

values. However, the `eval` primitive has never been widely adopted, in that process creation and lifecycle has always been dealt – in actually implemented systems – outside the tuple-based coordination system.

Starting the basic simple model presented above, different specific models can be conceived both for tuples (e.g., based on objects, records, logic predicates) and for pattern-matching mechanisms (e.g., object matching, data matching, logic unification). These differences are of little relevance to our discussion in this chapter, where we most focus on architectural and distributed software engineering issues.

3 Tuple-based Coordination and Mobility

Let us now review what are the main challenges that have to be addressed when developing distributed mobile applications, and let us show how tuple-based model can effectively deal with these challenges. To ground the discussion, an exemplary mobile computing application is introduced.

3.1 Mobile Computing Core Challenges

The core problems related to mobile computing derives from the fact that applications will be embedded in complex, open, dynamic and ever changing environments [EdwG01]. In particular:

- Agents are connected to each other via *dynamic wireless networks*. Apart from technological issues, what really matters is that these networks will be ever changing. Components will be dynamically added or removed from them. Their topology will change because of nodes mobility. An agent, executing in such a network, would perceive an ever-changing environment, running in different places and different contexts. Available communication partners can become unreachable in a matter of few seconds and new ones can show up.
- Besides being dynamic, these networks will be extremely *heterogeneous* and *huge*. Consider a scenario a few years hence in which a large city like Boston might have several wireless base stations in every building – a number of nodes in the order of 10^7. If most of the electrical devices in the buildings and those carried on by people are wirelessly networked too, then the total number of nodes could be as high as 10^{10}. If these nodes communicate peer-to-peer with nearby devices, then one could envision the entire city as connected into a mobile ad-hoc network approximately 10^3 hops in diameter.

- Since pervasive and mobile computing system will be everywhere and will have an impact on every moment of our life, characteristics like *security* and *robustness* will become even more important. Hackers will try to enter our cell-smart-phones and viruses will not let our cars brake.
- Even worse, these systems are inherently *difficult to test and debug*. Emergent unexpected situations can arise only when the system is actually deployed and off-line simulations can lead to wrong previsions. Moreover, in a dynamic system where components are mobile and wirelessly interacting, debugging is extremely difficult [EdwG01]: who is talking with whom? What happened in the past?

Other than mere technological issues, the above are mostly modeling and conceptual issues, impacting on the software engineering principles upon which to rely for mobile application development [ZamG05].

3.2 Why Tuple-based Coordination Models

Keeping in mind the above issues, the reasons why tuple-based coordination models (although originally conceived for parallel and concurrent systems) have revealed suitable for developing open, distributed and mobile applications too [CabLZ00], can be summarized as follows:

- *Uncoupling*. The use of a tuple space as the coordination medium uncouples the coordinating components both in space and time: an agent can perform an `out` operation independently of the presence or even the existence of the retrieving agent, and can terminate its execution before such a tuple is actually retrieved. Moreover, since agents do not have to be in the same place to interact, the tuple space helps to abstract from locality issues. In a scenario, such as mobile computing, where agents can come and go at any time, and can be at any location in a possible large network, the uncoupling feature is dramatically important.
- *Associative addressing*. The template used to retrieve a tuple specifies what kind of tuple is requested, rather than which tuple. This well suits mobile agent scenarios: in a wide and dynamic environment, a complete and updated knowledge of execution environments and of other application agents may be difficult or even impossible to acquire. As agents would somehow require pattern-matching mechanisms to deal with uncertainty, dynamicity and heterogeneity (as intrinsically exhibited by mobile computing scenarios), it is worthwhile integrating these

mechanisms directly in the coordination model, to simplify agent programming and to reduce application complexity.

- *Context Awareness*. A tuple space can act as a natural repository of contextual information, to let agents get access to information about the "what's happening" in the surrounding operational environment.
- *Security and robustness*. A tuple space can be made in charge of controlling all interactions performed via tuples, independently of the identity of involved agents. This fact, together with the simplicity of the model, increases the degree of robustness of ant systems based on such coordination model, and primarily of mobile computing systems.
- *Separation of concerns*. Coordination languages focus on the issue of coordination only: they are not influenced by characteristics of the host programming language or of the involved hardware architecture. This leads to a clearer coordination model, simplifies programming, and intrinsically suit open and dynamic scenarios.

Summarizing, the Linda coordination model grants the flexibility and the adaptability needed in developing applications in mobile computing scenarios.

3.3 A Case Study Application

To fix ideas, it may be of some use to introduce a concrete application case study. Exemplary for a wider range of mobile computing applications, let us consider a system to support tourists – each assumed to carry on a mobile device – in visiting a huge museum.

The devices carried on by the users can be exploited for helping tourists achieve goals such as retrieving information about art pieces, effectively orientating themselves in the museum, and meeting with each other (in the case of organized groups). Specifically, two specific representative problems can be: *(i)* to gather and exploit information related to art pieces they want to see; *(ii)* planning and coordinating their movements with other, possible unknown, tourists (e.g. to avoid crowd or queues, or to meet together at a suitable location).

To this end, we can assume that: *(i)* tourists are provided with a software agent running on some wireless handheld device, like a palm computer or a cellular phone, in charge of giving her/him information on art pieces and suggestions on where and when to move; *(ii)* the museum is provided with an adequate

embedded network infrastructure based on tuple-based coordination models. In particular, embedded in the museum walls (either associated to each artistic items or to each museum room), there will be a network of computer hosts, each capable of communicating with each other and with the mobile devices located in its proximity via the use of a short-range wireless link. Within such an infrastructure, a multiplicity of tuple spaces (e.g., one per museum room, plus any other ones which may be needed for administrative reasons) are made available to agents to interact with each other and to retrieve museum information; *(iii)* both the devices and the embedded hosts are provided with a localization mechanism to find out where they are actually located in the museum.

Despite of this coarse description we think that this kind of system is a case study that captures in a powerful way features and constraints of mobile computing system:

- It represents a very dynamic scenario: the system has to cope with different museum floor plans, a variable number of tourists following different schedules. Tourists entering and exiting the museum, and that possibly ignore or misunderstood their PDA's advices, etc. The uncoupling of tuple-based coordination model make these simply become non-issues.
- In huge museums there can be thousands of embedded electronic devices and people with mobile devices. There can be multiple systems concurrently running within the museum computer infrastructure (e.g. light and heating control systems) and other systems connected to these other services. The associative mechanism of tuple spaces helps in managing the right information without imposing a rigid schema on application agents and on the infrastructure.
- Agents (i.e., tourists) have the primary goal of discovery what's in the museum, i.e., to achieve context awareness. Tuple spaces may naturally be assumed to be a digital representation of a museum room, via which to acquire information about the context.
- The system should be secure and robust: malicious or bad-programmed agents could try to penetrate the system; embedded hosts can broke down, wireless network can have glitches, and there can be any kind of unexpected situations. The system can cope with these anomalies by controlling the requests posted in the form of tuples for security's sake, and redirecting the requests to other tuple spaces in a flexible way when part(s) of the systems are not available.
- By managing all interactions via of tuple spaces, security and monitoring rules can be defined and enforced separately from the logic of the museum services.

The above scenario and the associated coordination problems are of a very general nature, being isomorphic to scenarios such as, e.g., traffic management and forklifts activity in a warehouse, where navigators' equipped vehicles hint their pilots on what to do; or software agents exploring the Web, where mobile software agents coordinate distributed researches on various Web-sites. Therefore, also all our considerations are of a more general validity, besides the addressed case study.

4 Middleware Taxonomy

Given the above mentioned advantages of tuple-based models for mobile computing scenarios, it is not surprising that a variety of middleware infrastructures and services relying on tuple-based coordination models have been recently proposed. However, while committing to the same general ideas, these systems tend to focus on different aspects of the aforementioned problems and consequently adopt very different architectural solutions.

In order to study and compare such different systems, it is very important: *(i)* to focus on a specific comparable subset of the services offered by different middleware systems; and *(ii)* to produce an effective taxonomy on which to ground the comparison.

With regard to the former point, we will focus on those services supporting agents' coordination form a software engineering perspective. In particular, from such perspective, there are two fundamental building blocks at the base of every coordination activities that have to be supported: interaction mechanisms and context-awareness. On the one hand, it is obvious that coordination requires some form of interaction: agents need to communicate someway in order to decide/plan/synchronize their actions. On the other hand, coordination requires context-awareness by its own nature. An agent can meaningfully work together and combine efforts with other agents only if it is somehow aware of "what is around", i.e., its context. Of course, these two building blocks are tightly interwoven in that contextual information can be communicated only via the available interaction mechanisms. As already outlined, tuple-based models are particularly effective in supporting both these two activities. In fact, tuple-spaces indeed provide both an uncoupled communication mechanism and a repository for contextual information. Still, the specifics of different architectural solutions carry on different advantages and drawbacks.

Thus, with regard to the latter point, our proposal is to classify middleware infrastructures along three main axes (see figure 1):

1. Middleware Location
2. Communication Extent
3. Middleware Adaptability

And, on this base, to analyze how the positioning of a specific proposal along each of these axis may impact on agents' interaction mechanisms and agents' context awareness.

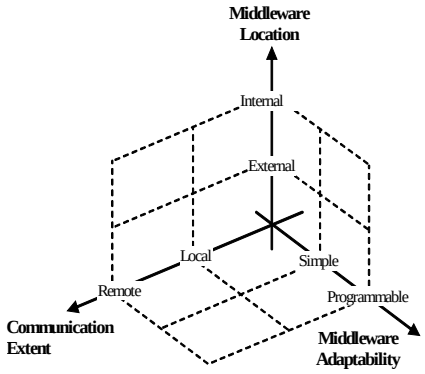

Fig 1. **The middleware 3D taxonomy space.**

4.1 Middleware Location

In this direction we want to classify various middleware infrastructures, by answering the following question:

Is the middleware something external, to which agents connect, or is something internal, like an agent sub-system (i.e. API)?

Here we have two landmark answers (landmark in the sense that also intermediate-borderline systems can be conceived):

- **External:** The middleware is something like an external service accessed by the agents. For example, a middleware server offering a shared data space to which agents can post and retrieve data would belong to this category.
- **Internal:** The middleware is strictly local and each agent is provided with its own private instance of the middleware. For example, a middleware API wrapping and enriching the agent communication features, would belong to this category.

External Middleware: Impact on interaction mechanisms and Context Awareness

The middleware locality strongly influences how and with whom agents interact, and consequently how contextual information is exchanged. Let us focus on the external middleware case, by considering the case study application. With this regard, we can suppose that the museum building is provided with a network of middleware servers, installed in every museum's rooms and providing suitable services to enable agents' interactions. Agents connect to the closest middleware server in order to interact, e.g. by posting and retrieving messages or subscribing to events (See Figure 2).

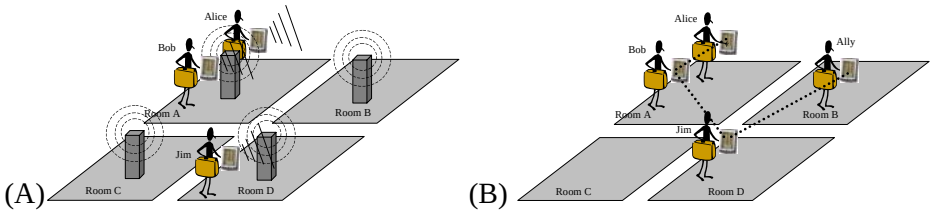

Fig 2. **(A) museum with a set of external middleware servers accessed by agents on a location basis. (B) we can suppose that users' agents are networked (e.g. via a mobile ad-hoc network) and they interact directly with each other by means of an internal middleware.**

The first obvious consideration we can make is that – by definition - this kind of middleware requires a deployed infrastructure. One the one hand, this means that this kind of middleware is not suitable in ad-hoc situations, where agents' coordination (interaction plus context-awareness), has to be realized in not-instrumented environments. On the other hand, the availability of an infrastructure can greatly reduce the agents (and thus the PDAs batteries) load, in that agents can demand to the infrastructure lots of operations. For example, the tuples' pattern matching operations can be performed on the infrastructure without consuming PDA resources..

Another important consideration is that an external middleware, possibly accessed by a vast number of agents, can be the source of scalability and robustness problems, representing a candidate bottleneck and single point of failure. If the middleware server in a room breaks down, the communication in all that room would be disrupted. However, an external middleware - especially if relying on a fixed network infrastructure – avoids most of resource constraints of mobile and pervasive devices and thus is less subject to hardware problems (e.g. run out of battery, wireless network glitches, just to name few examples).

Moreover, from the middleware developer point of view, an external middleware is inherently simpler that an internal one. External middleware, being detached from agents, does not require to manage mobility and consequent reconfigurations and dynamics of its own instances.

Internal Middleware: Impact on interaction mechanisms and Context Awareness

Turning our attention to the internal middleware case, we can suppose that the users' PDAs connect in a mobile ad hoc network enabling agents to communicate (see Figure 1). In this scenario most of the consideration made before are turned upside-down: an internal middleware is suitable to coordinate agents' activities in not-instrumented environments. It is thus cheaper to be realized, since it does not require installation costs. It is inherently scalable and robust, in that there are not bottlenecks or single points of failure. It is inherently more complex in that middleware instances – internal to the agents – need to react to agents' movements and to the consequent reconfigurations.

4.2 Communication Extent

In this direction we want to classify various middleware infrastructures, by answering the following question:

Given a set of middleware instances (whether internal or external), how are they connected?

Here we can simply identify two landmark answers:

- **Local:** In these kinds of middleware, the various instances are only locally connected or are not connected at all. This means that either a middleware instance can communicate with only neighboring other middleware instances or it cannot communicate only with connected agents.
- **Remote:** These kinds of middleware enable long range, multi-hop, communication between its instances.

Local Communication: Impact on interaction mechanisms and Context Awareness

Let us consider again Figure 1, it is clear that Bob and Alice can interact by means of the services (e.g. shared space facility) offered by the middleware server installed on room A. But how can Bob and Jim interact? It is clear that, if the middleware realize a local communication across its instances, Bob and Jim

cannot interact with the abstractions promoted by the middleware (e.g. because they access to separate – disjoint – shared data spaces). If they want to communicate, they have to meet in a specific room and use that room's middleware to interact. This is not related to external middleware only, if in the internal middleware scenario of Figure 2, the middleware enables just 1-hop communication, Alice and Ally need to meet in order to communicate. Given that, it is rather easy to see that this kind of middleware intrinsically defines a locality-scope for agents' interactions.

Of course, the same locality scope applies to context awareness. In fact, each agent can know only about facts happening in its immediate neighborhood, in the hope that these will be the most relevant for its actual execution.

Although this strict locality in agents' interaction and perception can be a severe limitation, it is not necessarily "bad". Locality scope reduces the problem of information overloading and enable the system to better scale with its size.

In the museum application for example, having a local communication middleware means, on the one hand, that agents could receive information only related to art pieces present in the room where they are located. In most of the situations, this is not a problem since it is likely that users will request information about art pieces they are actually seeing. Moreover, this enables the system to scale better, in that an agent is not bothered with unnecessary information related to irrelevant, faraway items.

On the other hand, strict local communication can instead represent a big obstacle for the motion coordination task. If, for example, two users located on the opposite sides of the museum want to meet somewhere, they have lots of difficulties in doing this. Since they cannot interact, their only possibility is either to wander randomly or to exploit information previously published within their locality scope by other agents. If for example, one of the two users as it moves in the museum stores, in all the middleware instances it connects with, the tuple "I will be in room A, at 10 am". Another user can use this information to easily meet that person at 10 am.

Remote Communication: Impact on interaction mechanisms and Context Awareness

In this kind of middleware all the middleware instances can interact with each other. Considering figure 1, this would be the case where for example all the servers are networked and data entering in one server is automatically replicated in all the others. This of course would enable long range interactions. In such

middleware the locality scope for agent interactions is considerably weakened. On the one hand this increases the system flexibility: an agent can be informed of relevant information happening far away. On the other hand, can create scalability problems and information overloading. To this end, further methods to filter and reduce accessible information should be implemented.

In the museum application for example, multi-hop communication would enable tourists to access information about every art pieces in the museum from wherever. Moreover, it would allow motion coordination even between faraway agents that would be able to exchange messages, disregarding their actual position, to decide a common motion strategy. The problem of this approach relates to information overload and over consume of network bandwidth, tourists must be able to filter only relevant information and high-level constraints must be enforced to limit bandwidth usage.

4.3 Middleware Adaptability

In this direction we want to classify various middleware infrastructures, by answering the following question:

Is the middleware capable of supporting agents' computational activities by means of programmable behaviors?

Here we provide two landmark answers:

- **Simple:** The middleware is not able to support any computational activity. All the computations are left to the agents. This kind of middleware provides a predefined set of capabilities implemented in a fixed way, without the possibility for the middleware itself or for an agent to change or customize middleware features.
- **Programmable:** is a system that is able to dynamically download, store and execute foreign code. Agents can thus program the middleware, not only by reshaping its predefined set of features, but also by implanting new programs and services. These new implanted services can be associated with some triggering conditions, to let the middleware execute those procedures whenever the proper conditions are met.

Simple Middleware: Impact on interaction mechanisms and Context Awareness

A simple middleware cannot adapt (or be adapted) to changing situations and provide agents only with a fixed set of unchangeable tools. Typically, a simple middleware enables plain communication between agents. With this regard, agents can exchange string-like messages or method invocations.

In the museum application for example, it could be desirable to adapt tourist information to PDAs' displays and users' settings. However, simple middleware could not flexibly adapt their services to different types of users' PDAs and such an operation could be done only in a static way. The middleware could be able to manage a predefined set of devices' profiles only.

Moreover, the fact that the middleware supports only string-based communication or method invocation (see later for other options) can be a constraint in some applications. For example, representing contextual information by means of plain strings can be not expressive enough, and can force agents to execute complex algorithms to understand it and decide what to do. For example, although the knowledge of all the agents' coordinates in the museum would be a complete contextual information for motion coordination tasks, it would be still difficult for an agent to decide what to do (i.e. where to go) on the basis of such rough information.

Programmable Middleware: Impact on interaction mechanisms and Context Awareness

A programmable middleware, on the contrary, by being able to store and execute foreign code is able to perform any kind of adaptation. Not only its mechanisms can be adapted to changing situation (e.g. the middleware could be programmed to automatically compress specified piece of data, depending on the available bandwidth), but taking the approach to the extreme, virus-like communication, based on mobile code can be enacted. Here the information being exchanged is embodied by mobile code, thus a message would be able to autonomously specify its routing, to automatically adapt and change its own content, and execute any kind of side-effect. Of course the flexibility of this kind of middleware, must be traded and balanced with the security issues, which naturally arise when possibly malicious code is allowed to run in the middleware

A programmable middleware offers much more flexibility. On the one hand, agents can program the middleware to let it filter and aggregate relevant contextual information [DayA99]. On the other hand, context information sources can describe their information not only by simple messages, but even by

complex programs. These programs can contain the algorithms on how to parse or interpret the contextual information or routing mechanism enabling also information fusion and aggregation [IntG00].

In the museum application for example, agents could be flexibly program the middleware to receive information suitable for their display capabilities. They could embed programs in the middleware to let it react to special events, in ways possibly not foreseen when the middleware had been first deployed. For example, the middleware could be programmed to block communication between a group of students' PDAs when their teacher has posted a question.

Finally, with regard to the motion coordination problem, we could imagine that an agent could send via the middleware something like a program (e.g. a routing algorithm) enabling other agents to reach a specific destination, by executing the received algorithm.

5 Current Middleware Infrastructures

In this section, we are going to survey different middleware infrastructure, under the previously made classification schema. Because the number of the proposed models and middleware is overwhelming and it is still growing very fast, we will try to give a classification based on an exploration of the introduced middleware space. For each type of middleware some relevant implementations will be presented. In figure 3, we depict the middleware that will be considered properly located in the taxonomy space

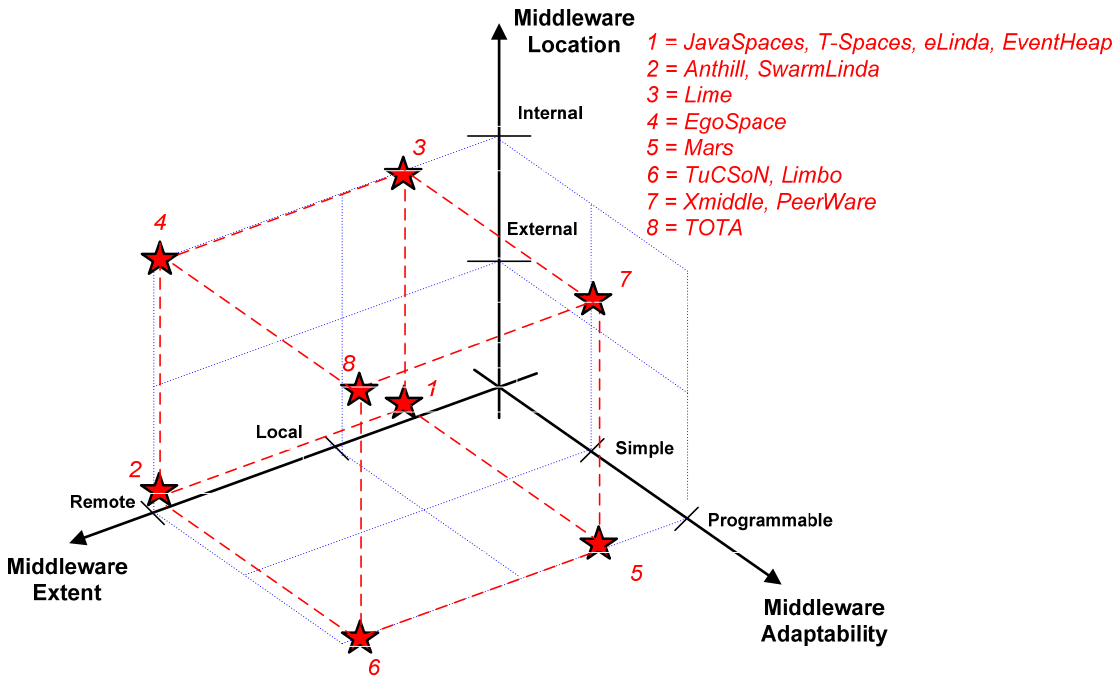

Fig 3. Surveyed middleware in the taxonomy space.

5.1 A Walk along the 'Communication Extent' Axis

We present the models and middleware populating our space by means of an imaginary 'walk' along the three dimensions constituting the taxonomy. Starting from considering the 'communication extent' axis, we can see that the taxonomy space is divided in two regions (see figure 4). In the rightmost region, local communication middleware defines boundaries either for agents' communication and context-awareness. This can turn out to be a problem, if agent need to acquire a global picture of the application-state, but it can also implicitly alleviate the problem of information overload and bandwidth over-exploitation. In the leftmost region, remote communication middleware do not impose boundaries in agents' communication thus increasing system's flexibility. However, if not properly controlled, this can lead to information overloading and exhaustion of network bandwidth.

In particular, we can consider those approach that are nearer to the origin of the axis, that means are *simple*, *external* and *local*. Here we can find the JavaSpaces [FreeHa99] proposal from SUN Microsystems, that consists in a specification of a framework for distributed network services to be implemented using the Java language. The JavaSpaces specification defines a tuple space where special tuples, instances of the Entry class, can be stored and retrieved, through the Java serialization mechanism. JavaSpaces can be

accessed directly addressing the tuple space instance, and exploiting its interface services, that includes a simple pattern matching mechanism. Even if *simple*, JavaSpaces provide a few interesting features, like the already mentioned tuple pattern matching mechanism, or the possibility to define a "leasing time" on a tuple (i.e., a time to live), support for tuple inserting notification and, last, the capability to extend the base tuple class with user defined data and services. A few interesting implementations of the JavaSpaces specification are GigaSpaces [GTS01], that also enables the access to the tuple space by mean of SOAP, and AutevoSpaces [Int03], that relies on a distributed tuple space implementation.

T-Spaces [WycML98] is the IBM answer to the JavaSpaces, and provides a Java implementation of a tuple space enhanced with both blocking and non-blocking tuple access services, rendez-vous specific services, indexing and support for tuple activity notification. Similarly to the above tuple spaces, T-Spaces must be directly addressed in order to exploit the provided services, even if several T-Spaces can be aggregated in order to build a "single space of spaces".

Another interesting approach is e-Linda [Wel04], that provides a Java implementation of a tuple space with a programmable matching engine. The latter can provide specialized matching logics, such as, for example, the search for a minimum value in the tuple space. Nevertheless, the matching engine cannot be dynamically programmed, that means the matching features must be known a priori.

Moving along the communication axis of the taxonomy space, we find the approaches that are catalogued as *simple*, *external* and *remote*. These systems can interact each other, that means that different instances can exchange data and information. Often, this is reached through a peer-to-peer network, where different hosts run an instance of the middleware, and such instances connects to other instances running on other hosts. An interesting approach in this direction is represented by SwarmLinda [ChaMT04], a Java implementation that exploits XML documents to describe tuples. SwarmLinda bases on concepts of swarm intelligence and multi agent systems modeling the tuple space as a set of nodes, and provided services (inserting, retrieving, etc.) performed by ants that travel across the nodes and search for (or carry on) one or more tuples. The interesting feature of SwarmLinda is its tuple aggregation, based on patterns criteria, thus similar tuples will be close together and kept (possibly) in the same node space. This implies that, while the whole system can be seen as a composition of distributed tuple spaces, it is really a single tuple space with clients connected to different instances but that perceive the system as unique.

Another interesting approach, quite similar to SwarmLinda, is Anthill [BabMM01], a middleware which relies on the Java implementation of JXTA [JXTA] and provides a self-organizing network of interconnected units, called “nests”, visited by ants, a kind of agents assigned to one or more task (e.g., tuple inserting, tuple retrieving, etc.). It is important to note that, ants, cannot communicate together directly, but must leave information that can be exploited by other ants (this kind of indirect communication is called stigmergy).

Another middleware is EventHeap [JohF02], where tuples (called events) can be made of fields which value is not known a priori (post fields) or known but not relevant (virtual fields), thus the exact behavior of the tuple space will depend on the value assigned later to those fields in the tuples. EventHeap has been implemented firstly on top of T-Spaces, then it has been rewritten in Java starting from scratch.

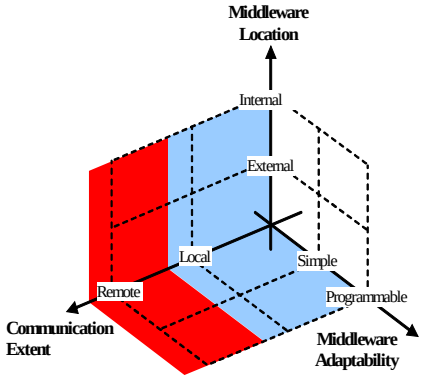

Fig 4. Different zones along the communication extent axis constrain the application design.

5.2 A Walk along the ‘Location’ Axis

Turning our attention to the ‘location’ axis, we can find other two regions (see figure 5). In the bottom one, where External middleware resides, we see infrastructures that have problems in a MANET scenario, because they require an underling common infrastructure. Moreover these infrastructures can induce scalability and robustness problems since the middleware can in principle be accessed by a great number of agents thus resulting in a bottleneck or in a single point of failure. The top region characterized by Internal middleware can be painlessly applied in a MANET scenario, because such models and middleware don’t require a common infrastructure accessible by different agents. Moreover, internal middleware infrastructures tend to scale with the size of the system, since they are replicated in every agent and for the same reason they are robust in that they tend to avoid single points of failure.

Lime (LInda in a Mobile Environment) [PicMR01] is an *internal* middleware that is mapped also as *simple* and *local*. The key idea of Lime is that each mobile entity, either a software agent or a physical device, is associated to a personal tuple space, accessed through a Interface Tuple Space (ITS). When mobile entities meet together, their ITS are transparently merged, in order to enable coordination. In other words, each mobile entity performs tuple operation over its personal tuple space, and the latter is "updated" with other personal tuple space information as possible. It is important to note that it is possible to define private tuple spaces, that will not be exchanged with other mobile entities; moreover Lime supports reactivity, that is the capability to perform a specific operation when a specific tuple is found in the tuple space.

EgoSpace [RomJ02] is a *simple*, *internal* and *remote* middleware that connects each entity belonging to the network and running a middleware instance. In this way, a distributed and collaborating architecture can be build and information (i.e., tuples) can be scattered across the instances. An important feature of EgoSpace is the capability to express the interest level for information belonging to specific geographical areas, thus it is possible to define boundaries for the search for and retrieving of tuples, saving also the bandwidth.

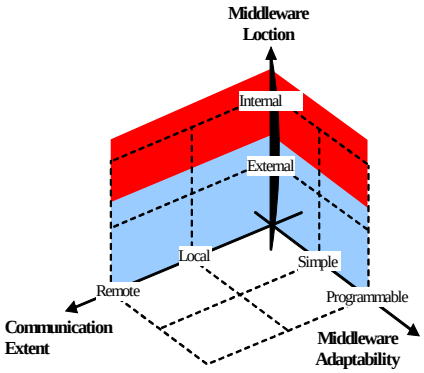

Fig 5. **Different zones along the middleware location axis constrain the application design.**

5.3 A Walk along the Adaptability Axis

The final dimension we follow is the one defining the middleware adaptability (see figure 6). Here, we can define a leftmost region characterized by Simple (i.e. not programmable) middleware infrastructures that are easier to be used since they exhibit only a fixed set of capabilities. The drawback of this simplicity is that they are best suited only to relatively static scenarios, because their fixed set of capabilities cannot be

customized to dynamics and unexpected situations. On the contrary, in the rightmost region, Programmable middleware infrastructures are extremely flexible and well suited for dynamic application scenario. The drawback of all this flexibility is that they tend to be more complex to be used and they can introduce security concerns by offering the possibility to install an run foreign – possibly malicious – code.

Focusing on these latter, we can consider those models that are *programmable*, *external* and *local*. A first interesting approach is MARS (Mobile Agent Reactive Spaces) [CabLZ00], that extends the Sun JavaSpaces. MARS defines a set of independent spaces, each one tied to an host (i.e., a local execution environment), that agents can exploit in order to perform tuple operations, accessing through the MARS interface. Each MARS instance can support reactivity, that is the capability of perform some operation when a tuple operation (e.g., inserting, reading, extracting) is performed. To do this, MARS exploits a meta tuple space (meta-space) that stores reactions to be performed as tuples. Each time a tuple operation is performed, the meta-space is searched for a reaction tuple and, if the latter is found, the reaction is executed.

Now it is time to have a look to those middleware that are *programmable*, *external* but *remote*, thus that are able to make their instances communicating. A representative example is the TuCSoN system [OmiZ99], focussing on coordination of mobile agents. TuCSoN defines the concept of tuple centre, an instance running on an internet host and that can be connected to other tuple centers. Tuple centers are uniquely named across internet, thus agents can directly interact with each one of them, if they are network aware, or can interact with the local tuple space that will interact transparently with the others if agents are not network aware. It is important to note that a tuple centre is not only a tuple space, since it can support specification tuples, that defines the reaction logic to communicative actions on the tuple space. A different approach is that followed by Limbo [DavWFB97], that defines a set of tuple spaces interacting each other, but where the programmable feature is reached sub-typing an existing tuple space. In other words, once a client wants to specify the behavior of the tuple space, it has to place a few tuples describing that behavior in the tuple space, and then to place a special “create tuple space” tuple, that will be handled by the tuple space itself, that will fork into a new tuple space with the specified behavior.

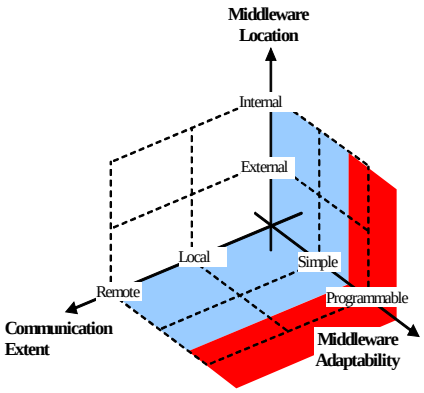

Fig 6. **Different zones along the adaptability axis constrain the application design.**

5.4 Other Programmable and Internal Middleware

Changing the direction of the middleware location, we can survey on *programmable*, *internal* and *local* middleware. An interesting project, in this field, is represented by XMiddle [MasCE01], that proposes a tuple spaces based on XML trees, where each mobile entity carries on its own tree, that is merged with other trees when agents meet together. Thanks to the exploitation of the XML language, XMiddle can share tailored portion of data, depending on the differences between spaces that are synchronizing. Most important, the use of XML allows XMiddle to handle structured data, thus it is able to associate a semantic to the exchanged data, allowing the implementation of different synchronization/reconciliation protocols.

Another interesting approach is PeerWare [CugP], where again each node runs a middleware instance handling its local data, and merging the data each time it joins other peers. It is important to note that this approach explicitly recognize the local space and the global space (the one made by all the local spaces available on-line), defining different primitives for both the spaces. Furthermore, a special set of primitives are available for both the spaces, in order to define the programmability of the space.

The last region of the taxonomy left out by our imaginary travel, is that which defines *programmable*, *internal* and *remote* middleware; here we can find the TOTA [MamZ04] approach. The key idea behind TOTA is that a tuple must be propagated in the surrounding environment following a specific rule; in other words the information is composed by the tuple itself and the rules to be applied on the tuple. Once the tuple is outputted, it is distributed among each node (running the TOTA middleware), meaning that the tuple can be copied as it is or it can be modified in order to reflect changes in the environment.

6 Open Issues and Research Directions

A number of apparently very diverse research areas, i.e., P2P overlay networks, swarm intelligent systems, pervasive and ubiquitous computing systems, share indeed some common issues with tuple-based coordination models and middleware. Still, the relations and synergies between these areas are mostly unexplored, thus representing fertile ground for further investigations.

6.1 Overlay Networks and Overlay Data Structures

Overlay networks can be defined as routing distributed data structures providing agents with a suitable application-specific view of the network (i.e. they allow agents to perceive a specific overlay topology of the network) [Rat02, RaoP03]. These structures are typically created by deploying across the network suitable routing information and are at the basis of a number of mobile computing scenarios.

In a lot of applications, in fact, the utility of a network of mobile computing devices derives primarily from the data and information it holds. The identity of the individual nodes storing the data tends to be less relevant. Think, for example, at sensor network applications or at file-sharing applications between mobile devices. Accordingly, suitable interaction models and communication abstractions should be flexible and application-tailored and not tied to the identities of the individual components. In this context, overlay networks can offer several application-specific mechanisms to route information in a dynamic network:

- Location-based routing, where an agent takes advantage of location information to access resources within suitable locality constraints (e.g. find all the printers on this floor, or find the closest gas station);
- Content-based routing, where data are searched and accessed on the basis of their content rather than on the basis of nodes' network addresses or location. For example, in a mobile sensor network scenario [Rat02] an agent may be interested to the occurrence of the data named "truck sightings". The network must provide means to effectively access these data wherever they happened.

Tuple based modes are a particularly fertile ground to investigate in order to develop much more advances kinds of overlay networks. For instance, a set of tuple spaces networked with each other can naturally lead to a semantically-enriched overlay network, from which to retrieve data in a more meaningful

way than in current approaches. Moreover, programmable tuple spaces like MARS [CabLZ00] and TOTA [MamZ04] can naturally support the reconfiguration, updating, and maintenance, of such overlay networks.

Overlay data structures can generalize overlay networks by encoding and providing agents with several pictures, possibly locally confined, of specific aspects of the agents' operational environment. They are not focused on the network topology only, but they are general purpose. Agents can access overlay data structures to achieve awareness and possibly modify specific contextual aspects. The strength of these overlay data structures is that they can be accessed piecewise as the application agents visit different places of the distributed environment. This lets the agents to access the right information at the right location. Again, tuple spaces are a perfect abstraction on which to build such overlays.

6.2 Stigmergy and Swarm Intelligence

Overlay data structures realized on top of tuple-based models can naturally accommodate stigmergic interaction patterns. These patterns are at the core of “swam intelligent” systems [BonDT99], i.e., systems where a large number of simple agents coordinate (often mimicking natural and biological system) in an indirect way, via sensing of digital pheromones in a virtual environment, to achieve – in an adaptive and self-organizing way – tasks that far exceed their capabilities as single individuals [Par04].

Tuple based models are naturally suited at supporting these interaction patterns, in that tuple spaces can be used to store the pheromones at the basis of stigmergy interactions. Moreover, active tuple spaces can naturally provide functionalities to let pheromoes (implemented by means of tuples) to change on needs (e.g. to diffuse across the network, to be aggregated and combined, to evaporate if not used and reinforced, etc.).

It is rather natural to see that in such innovative scenarios there are endless research open directions with regard to tuple-based coordination models. How can tuple based models be employed to control and govern self-organizing stigmergic coordination activities? How can tuple spaces fill the semantic gap between heterogeneous agents that coordinate my means of tuples and “ant-like” agents that simply coordinate by reacting upon pheromone sensing? What applications can be enabled by such rich coordination models? We do not have any answer ready, but really would like to have some.

6.3 Pervasive Spaces and Tuple Spaces

Several recent researches in the area of pervasive and ubiquitous computing suggest that, to enable spontaneous interactions among a set of computer-based devices embedded in an environment (e.g., smart rooms, smart furniture, smart objects) and also to support advanced interaction models between users and surrounding (computer-enriched) environments, the environment itself should reify in some sort of digital abstractions.

The above consideration led to several proposals for middleware infrastructure based on the concept of "active spaces" [Rom02]. Active spaces are sorts of digital representation of a physical environment, where each resource in the environment (a user, a computer, as well as any computer-based device and any computer-based object) has a digital representation, and is provided with mechanisms to interact with the other resources. Thus, active spaces act as sorts of shared data spaces, where high-level interaction patterns can be promoted and from where high-level contextual information can be obtained.

How and to which extent tuple-based coordination models and active-spaces-based approaches can be made co-exists and converge into a single coordination model, supporting both messaging, synchronization, exchange of raw data and of semantic high-level information, is still to be investigated.

Strictly related, it is still to be investigated how and to which extent emerging pervasive computing technologies can be used to conceive brand new architectural solutions for tuple-based middleware models. For example, one could think at exploiting the stable memory of RFID tags to deploy – in a massively distributed way – tuples and tuple spaces in any physical environment [MamZ05].

7 Conclusions

To quickly conclude and summarize, in this chapter we have tried to outline how tuple-based models and middleware can indeed offer a valuable tool to support uncoupled agents' coordination and context-awareness in mobile computing scenarios. While the number of research proposals in the area already testifies such fact, we hope that the introduced taxonomy and the critical survey of existing systems can be of some help to reach a better understanding of the software engineering issues involved, and to properly direct developers towards the adoption of specific tuple-based middleware systems suitable to their purposes.

In any case, despite this assessed suitability of tuple-based coordination models for mobile computing, developers and researchers should also be aware that there are a number of fascinating research issues worth to be investigated.

References

[BabMM01] O. Babaoglu, H. Meling, A. Montresor, “Anthill: a Framework for the Design and the Analysis of Peer-to-Peer Systems”, 4th European Research Seminar on Advances in Distributed Systems (ERSADS '01), Bertinoro, Italy, 2001

[BonDT99] E. Bonabeau, M. Dorigo, G. Theraulaz, “Swarm Intelligence”, Oxford University Press, 1999.

[Bor02] C. Borcea, et al., “Cooperative Computing for Distributed Embedded Systems”, 22th International Conference on Distributed Computing Systems, Vienna (A), IEEE CS Press July 2002.

[CabLMZ03] Giacomo Cabri, Letizia Leonardi, Marco Mamei, Franco Zambonelli, “Location-dependent Services for Mobile Users”, IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems And Humans, 33(6):667-681, Nov. 2003

[CabLZ00] G. Cabri, L. Leonardi, F. Zambonelli,, “MARS: A Programmable Coordination Architecture for Mobile Agents”, IEEE Internet Computing, 4(4):26-35, July-August 2000

[ChaMT04] A. Charles, R. Menezes, R. Tolksdorf, “On the Implementation of SwarmLinda”, in the Proceedings of the 2004 ACM Southeastern Conference, March 2004.

[CugP] G. Cugola, G. P. Picco, "PeerWare: Core Middleware Support for Peer-To-Peer and Mobile Systems", Technical report available at the URL: http://peerware.sourceforge.net/

[DavWFB97] N. Davies, S. Wade, A. Friday, G. Blair, "Limbo: A Tuple Space Based Platform for Adaptive Mobile Applications", in the Proceedings of the International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP '97), Toronto, Canada, 1997

[DayA99] A. Dey and G. Abowd, “The Context-Toolkit: Aiding the Development of Context-Aware Applications”, Proc. Conf. Human Factors in Computing Systems (CHI), ACM Press, New York, 1999, pp. 434–441.

[EdwG01] K. Edwards, R. Grinter, “At Home with Ubiquitous Computing: Seven Challenges”, Ubicomp 2001.

[EstC02] D. Estrin, D. Culler, K. Pister, G. Sukjatme, “Connecting the Physical World with Pervasive Networks”, IEEE Pervasive Computing, 1(1):59-69, Jan. 2002.

[FreeHA99] E. Freeman, S. Hupfer, K. Arnold, “JavaSpaces Principles, Patterns and Practice”, Addison Wesley, 1999

[GelC92] D. Gelernter, N. Carriero, "Coordination Languages and Their Significance", Communications of the ACM, 35(2):96-107, Feb. 1992.

[GST01] GigaSpaces Technologies Ltd., available material at the URL htpp://www.gigaspaces.com/index.html, 2001

[Int03] Intamission Ltd., “AutevoSpaces Product Overview”, available at the URL http://wwww.intramission.com/downloads/datasheets/AutevoSpaces-Overview.pdf, 2003

[IntG00] C. Intanagonwiwat, R. Govindan, D. Estrin, “Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks”, in the Proceedings of the ACM MobiCom Conference, Oct. 2000.

[JohF02] B. Johanson, A. Fox, “The Event Heap: A Coordination Infrastructure for Interactive Workspaces”, in the Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2002), Callicoon, NY, USA, 2002

[JXTA] The JXTA project, main web site: http://www.jxta.org

[MamZ04] M. Mamei, F Zambonelli, “Programming Pervasive and Mobile Computing Applications with the TOTA Middleware”, in the Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communication (Percom2004), Orlando (FL), 2004

[MamZ05] M. Mamei, F Zambonelli, “Spreading Pheromones in Everyday Environment Through RFID Technology”, in the Proceedings of the 2nd IEEE Symposium on Swarm Intelligence, Pasadena (CA), 2005

[MasCE01] C. Mascolo, L. Capra, W. Emmerich, “An XML based Middleware for Peer-to-Peer Computing”, 1st IEEE International Conference of Peer-to-Peer Computing, Linkoping (S), 2001

[OmiZ99] A. Omicini, F. Zambonelli, “Coordination for Internet Application Development”, Journal of Autonomous Agents and Multi-Agent Systems, 2(3):251–269, Sept. 1999.

[Par04] V. Parunak, S. Brueckner, J. Sauter, “Digital Pheromones for Coordination of Unmanned Vehicles”, Workshop on Environments for Multi-agent Systems (E4MAS), LNAI 3374, Springer Verlag, 2004.

[PicMR01] G. P. Picco, A. L. Murphy, G. C. Roman, “LIME: a Middleware for Logical and Physical Mobility”, In Proceedings of the 21st International Conference on Distributed Computing Systems, IEEE CS Press, July 2001.

[RaoP03] A. Rao, C. Papadimitriou, S. Ratnasamy, S. Shenker, I. Stoica, “Geographic Routing without Location Information”, in the Proceedings of the ACM Mobicom Conference, San Diego (CA), Oct. 2003.

[Rat02] S. Ratsanamy, et al., “GHT: A Geographic Hash Table for Data-Centric Storage”, Int. Workshop on Wireless Sensor Networks and Applications, Atlanta, 2002.

[RomJ02] G. C. Roman, C. Julien, Q. Huang, “Network Abstractions for Context-Aware Mobile Computing”, ICSE ‘02, Orlando (FL), ACM Press, May 2002

[Rom02] M. Roman et al., “ Gaia : A Middleware Infrastructure for Active Spaces”, IEEE Pervasive Computing, 1(4):74-83, Oct.-Dec. 2002.

[Wel04] G. C. Wells, “New and Improved: Linda in Java”, in the proceedings of the Third International Conference on Principles and Practice of Programming Java (PPPJ), Las Vegas, Nevada U.S.A., 2004

[WycML98] P. Wyckoff, S. W. McLaughry, T. J. Lehman, D. A. Ford, “T Spaces”, IBM Systems Journal, 37(3):454-474, 1998.

[ZamG05] F. Zambonelli, M.P. Gleizes, M. Mamei, R. Tolksdorf, Spray Computers: Explorations in Self organization“, Journal of Pervasive and Mobile Computing, 1(1):1-20, May 2005.

BIOGRAHIES OF THE AUTHORS

Giacomo Cabri is a research associate in Computer Science at the University of Modena and Reggio Emilia, since 2001. He received the Laurea degree in Electronic Engineering from the University of Bologna in 1995, and the PhD in Computer Science from the University of Modena and Reggio Emilia in 2000. His research interests include methodologies, tools and environments for agents and mobile computing, wide-scale network applications, and object-oriented programming.

Luca Ferrari is a PhD student in Computer Science at the University of Modena and Reggio Emilia. He received the Laurea degree in Computer Science Engineering from the University of Modena and Reggio Emilia in 2002. His research activity covers the study of models, methodologies and environments for mobile agent systems, object-oriented programming and Java-based technologies.

Letizia Leonardi is a full professor in Computer Science at the University of Modena and Reggio Emilia, since 2001, where she teaches basic and advanced computer science courses. She received the Laurea degree in Electronic Engineering in 1982 and the PhD in Computer Science in 1989, both from the University of Bologna. Her research interests include design and implementation of coordination infrastructures for mobile agent systems, object-oriented programming environments, and parallelism and distribution issues, especially as they apply to object systems.

Marco Mamei is research associate at the University of Modena and Reggio Emilia, since 2004. He obtained the Laurea degree in Computer Science in 2001, and the PhD in Computer Science in 2004, both from the University of Modena and Reggio Emilia. His current research interests include distributed and pervasive computing, swarm intelligence, and multiagent systems. He is a member of the IEEE, AIIA, and TABOO.

Franco Zambonelli is professor in Computer Science at the University of Modena and Reggio Emilia, since 2001. He obtained the Laurea degree in Electronic Engineering in 1992, and the PhD in Computer Science in 1997, both from the University of Bologna. His current research interests include: distributed and pervasive computing, multiagent systems, agent-oriented software engineering. He is a member of IEEE, ACM, AIIA, and TABOO, and founding member of the "Autonomic Communication Forum".